KIV/CPP – Programování v jazyce C++

06. Copy a move sémantika, šablony

Martin Úbl

KIV ZČU

2023/2024

l-value, r-value

- *l-value*
- „locator value“, někdy „left-hand side value“
- všechny identifikátory, které označují konkrétní místo v paměti
- „z pohledu kódu“; fyzicky i jiné objekty mají nějaké místo v paměti, jen ho nejsme schopni identifikovat názvem
- do *l-value* jsme schopni přiřadit hodnotu
  - výjimka jsou `const` proměnné, ale i ty jsou *l-value*

```
int a = 5;          // a je l-value
std::string b;      // b je l-value
MyClass c;          // c je l-value
```

l-value, r-value

- *r-value*
- „right-hand side value“
- dočasné a přechodné objekty
- (syntakticky) nemají adresovatelnou paměť – nemají identifikátor
- nelze do nich přiřadit

```
int a = 5;
    // a je l-value, 5 je r-value
std::string b = "hi";
    // b je l-value, "hi" je r-value
float c = get_c();
    // get_c() je (vraci) r-value

5 = c; // ouch!
```

l-value, r-value

- *l-value* reference
- reference na *l-value*
- klasická reference, s jakou jsme pracovali
- nekonstantní nelze navázat na *r-value*

```
int a = 5;
int& ra = a; // ok

int& rb = 5; // ouch!
```

l-value, r-value

- konstantní *l-value* reference
- také reference na *l-value*
- při vázání na *r-value* díky `const` kompilátor převede na *l-value*

```
const int& rb = 5; // ok

void my_func(const int& a) {
   ...
}
...
my_func(a);  // ok
my_func(10); // ok
```

r-value reference

- *r-value* reference
- reference na *r-value*
- nelze vázat na *l-value*
- `std::move`
- platnost má jako běžná reference, jen nemusí být const pro *r-value*
- uvození &&
- dovoluje fungování move sémantiky

```
int&& rb = 5; // ok

void my_func(int&& a) {
   ...
}
my_func(a)    // ouch!
my_func(10); // ok
```

r-value reference

- *r-value* reference
- dovoluje rozlišit dočasné a „trvalé“ instance

```
void my_func(int&& a) {
  // pro docasne
}
void my_func(const int& a) {
  // pro trvale
}

my_func(a)   // ok
my_func(10); // ok
```

l-value a r-value kontext

- lze rozlišit kontext volání metody objektu
- jiné chování při volání metody nad *l-value*, než nad *r-value*
- uvození & pro *l-value* a && pro *r-value* za signaturou metody

```
class Test {
  public:
    std::string GetContext() & {
      return "l-value";
    }
    std::string GetContext() && {
      return "r-value";
    }
};

a.GetContext();       // l-value
Test().GetContext(); // r-value
```

„C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.“

- Bjarne Stroustrup

„There are only two kinds of languages: the ones people complain about and the ones nobody uses.“

- Bjarne Stroustrup

Copy a move sémantika

- životnost objektu by měla být ohraničena i sémanticky
- copy sémantika
  - „klonuje“ zdroj
  - při konstruování nebo přiřazením
- move sémantika
  - „přesouvá“ zdroj
  - při konstruování nebo přiřazením
- příklady:
  - `std::unique_ptr` – pouze move
  - `std::thread` – pouze move
  - `std::ostream` a `std::istream` – pouze move
  - STL kontejnery – move i copy
- spíše než sada exaktních pravidel jde o sémantické odlišení kontextu
- lze lépe vyjádřit, kdo jaký objekt vlastní

Copy a move sémantika

- copy konstruktor a copy assignment
- objekt lze kopírovat při vytváření jiného
- nebo přiřazením do již existujícího objektu
- nemodifikuje zdrojový objekt

```
// copy konstruktor
MyClass(const MyClass& o) : mValue(o.mValue) {
}

// copy assignment operator
MyClass& operator=(const MyClass& other) {
  mValue = other.mValue;
  return *this;
}
```

Copy a move sémantika

- move konstruktor a move assignment
- objekt lze přesunout při vytváření jiného
- nebo přiřazením do již existujícího objektu
- modifikuje zdrojový objekt („vyprazdňuje“ ho)

```
// move konstruktor
MyClass(MyClass&& o) : mValue(o.mValue) {
  o.mValue = 0;
}

// move assignment operator
MyClass& operator=(MyClass&& other) {
  mValue = other.mValue;
  other.mValue = 0;
  return *this;
}
```

Copy a move sémantika

- copy konstrukce

```
MyClass a(other);
```

- move konstrukce

```
MyClass b(std::move(other));
```

- copy assignment

```
a = other;
```

- move assignment

```
b = std::move(other);
```

Copy a move sémantika

- pokud máme v úmyslu vyjádřit vlastnictví, existují pravidla
- *Rule of zero*
  - nedefinujeme sami move/copy konstruktory a operátory, ani destruktor
  - využíváme existující třídy, co to dělají za nás
  - např. `std::unique_ptr`
- *Rule of five*
  - definujeme vše:
    - copy konstruktor
    - copy assignment operátor
    - move konstruktor
    - move assignment operátor
    - destruktor

„If you think it's simple, then you have misunderstood the problem.“

- Bjarne Stroustrup

Šablony

- šablony
- prvek generického programování
- dovoluje zobecnit implementaci pro více parametrů
- generování kódu v čase kompilace
- parametrizace:
  - typem
  - hodnotou
  - proměnným seznamem (tzv. *variadické šablony*)
- klíčové slovo `template`, `typename` a špičaté závorky
- šablonové funkce, metody, třídy, constexpr výrazy, ...

```
template<typename T>
void DoSomethingGeneric(const T& a) {
   ...
}
```

Šablony

- již jsme je potkali
  - `std::array<int, 5>`
  - `std::vector<int>`
  - `std::function<void()>`
  - ...
- tyto třídy jsou obecné
- předem neví, jaký typ mají obalovat, vnitřně reprezentovat
- rozhodne se až v momentě tzv. *specializace*
  - explicitní nebo implicitní uvedení šablonových parametrů
  - vygeneruje se kopie kódu pro danou sadu parametrů

```
std::array<int, 5> arr;
```

- specializuje se pro `int` a velikost `5`

Šablony

- parametrizace typem
- klíčové slovo `typename` nebo `class` (ekvivalentní)

```
template<typename T>
void PrintLine(const T& something) {
  std::cout << "Neco: " << something
            << std::endl;
}

PrintLine("Tree fiddy");
PrintLine(3.50);
PrintLine<int>(999);
PrintLine<Monster>(lochNess);
```

Šablony

- parametrizace hodnotou

```
template<size_t Dim>
class Vector {
  private:
    std::array<double, Dim> mCoords;

    ...
};
```

- typicky pro nějaké analyticky definované třídy, které se mohou lišit parametry
- změnou hodnoty se v podstatě vytvoří nový typ

Šablony

- parametrizace typem i hodnotou
- lze parametrizovat více typy i hodnotami a kombinovat

```
template<typename T, size_t Size>
std::array<T, Size> FillNatural() {

  std::array<T, Size> ret;
  std::generate(ret.begin(), ret.end(),
          [n = 1]() mutable { return n++; });

  return ret;
}

auto arr = FillNatural<long, 1000>();
```

Šablony

- generické typy
- např. zásobník (zjednodušeně) bez nutnosti společného předka
  - jako např. Java a `java.lang.Object`

```
template<typename T>
class Stack {
  private:
    std::vector<T> mData;

  public:
    void push(const T& value);
    T& top() const;
    void pop();

    bool empty() const;
};
```

Šablony

- explicitní specializace funkce/metody
- např. jedna výjimka z jinak generického chování

```
template<typename T>
void do_something(const T& value)
{
  //
}

template<>
void do_something<double>(const double& value)
{
  //
}
```

Šablony

- explicitní specializace metody
- např. jedna výjimka z jinak generického chování třídy

```
template<typename T>
class Test {
  public:
    void Do_This();
    void Do_That();
};

template<>
void Test<int>::Do_This() {
  //
}
```

Šablony

- každý typ, pro který se specializuje, musí podporovat vše, co se provádí v těle metody
  - např. inkrementace operátorem ++ – všechny typy musí mít definován operátor ++

```
template<typename T>
T& increment(T& val) {
  return val++;
}

int a = 5;
increment(a); // ok

std::string s{ "one␣does␣not␣simply␣increment"
               "␣a␣string" };
increment(s); // ouch!
```

Šablony

- podmíněná specializace
- `std::enable_if`
- dovoluje překladači udat podmínky, za kterých se má šablona specializovat
- podmínky, např.:
  - `std::is_integral`
  - `std::is_floating_point`
  - `std::is_signed`
  - `std::is_pod`
  - `std::is_class`
  - a další...

Šablony

- může být těžkopádné

```
template<typename T,
         typename std::enable_if<
                     std::is_integral<T>::value
                  >::type* = nullptr>
T& increment(T& val) {
  val++;
  return val;
}
```

- dá se trochu zkrátit
  - místo `std::enable_if<...>::type` psát `std::enable_if_t<...>`
  - místo `std::is_integral<T>::value` psát `std::is_integral_v<T>`

- stále těžkopádné

```
template<typename T,
         typename std::enable_if_t<
                      std::is_integral_v<T>
                  >* = nullptr>
T& increment(T& val) {
  val++;
  return val;
}
```

Šablony

- dá se trochu zpřehlednit pomocí *concepts*
- viz další přednášky

```
template<typename T>
concept TIntegral = std::is_integral_v<T>;

TIntegral auto& increment(TIntegral auto& val)
{
  val++;
  return val;
}
```

Šablony

- `constexpr` šablony

```
template<typename T>
constexpr T PI = T(3.14159265358979323L);
```

Šablony

- šablonové lambda funkce (od C++20)
- do lambda funkce lze vložit šablonovou definici za *capture* blok
- pravidla jsou pak v zásadě stejná jako u obyčejných funkcí

```
auto multiply_2 = []<typename T>(T a) -> T {
        return a + a;
};
```

- pozn.: pozor – čím více instancí šablony, tím více lambda funktorů a tedy mnohem „lokálnější“ bobtnání kódu

Šablony

- šablonové lambda funkce (od C++20)
- co nelze udělat generickou lambdou?

```
auto vf = []<typename T>(std::vector<T>& a) {
  // something
};
```

- lze takto třeba definovat šablonový parametr konkrétního typu a vynutit na vstupu `std::vector`

Šablony

- kuriozity
- např. faktoriál rekurzivní specializací šablon

```
template <int N>
struct Factorial {
  static const int value =
     N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
  static const int value = 1;
};
```

- není dobrý nápad – proč?

Šablony

- faktoriál rekurzivní specializací šablon
- není dobrý nápad:
  - protože faktoriál rekurzí nikdy nebude dobrý nápad (viz PPA a jiné)
  - pro `N = 10` se vytvoří 10 struktur s jedním prvkem
  - musí se vyhodnotit v čase kompilace – zpomalení kompilace
- jen demonstruje genericitu šablon
- technicky vzato je systém šablon turingovsky úplný

Šablony

- co dál?
- variadické šablony
- perfect forwarding
- např. `emplace_back`
  - in-place vložení prvku např. do vectoru
  - pak není třeba více kopií
  - nebo více `std::move`
- SFINAE
- concepts